Prototyping and Simulating Parallel, Distributed Computations with VISA

Isabelle M. Demeure and Gary J. Nutt

CU-CS-450-89 November 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309.

This research has been supported by NSF Cooperative Agreement DCR-8420944, NSF grant CCR-8802283 and AFOSR-85-0251

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **NOV 1989**	2. REPORT TYPE	3. DATES COVERED **00-00-1989 to 00-00-1989**

4. TITLE AND SUBTITLE **Prototyping and Simulating Parallel, Distributed Computations with VISA**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Department of Computer Science,University of Colorado,Boulder,Co,80309**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**27**	

Abstract

Designing high performance distributed computations is a challenging task. In this paper, we describe VISA (VISual Assistant), a software tool to support the design, prototyping, and simulation of parallel, distributed computations. In particular, VISA is meant to guide the choice of partitioning and communication strategies for such computations, based on their performance. VISA uses ParaDiGM (Parallel Distributed computation Graph Model) as a basis for its graphical interface. VISA supports the editing of ParaDiGM graphs, and the animation of these graphs to provide visual feedback during simulations. Summary results are available when a simulation terminates.

We introduce the ParaDiGM constructs and describe the functionality of VISA. We illustrate its utility by providing simulations of two computations under various load conditions.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

Contents

List of Figures

1 Introduction - Motivation

In recent years, because of decreasing hardware and interconnection costs, organizations have made large investments in networks of computers. Designing efficient distributed computations is a challenging task. One of the challenges lies in the *partitioning* activity which involves dividing a problem into units (which we will informally call *processes*) that can be distributed over the various processors of a distributed hardware environment. A second challenge lies in the choice of the amount and the nature of the communication and synchronization that have to take place among the processes, or *communication* strategy. The choice of the partitioning strategy constrains that of the communication strategy, and vice versa; in order to design an efficient distributed computation one has to choose a good combination of both strategies, or *partitioning and communication strategy.* The goal of this paper is to describe a software tool called VISA (VISual Assistant) that is designed to support prototyping and simulation of distributed computations, and to illustrate how VISA can be used to experiment with alternative partitioning and communication strategies for the same problem, in order to design efficient distributed computations.

The computations of interest here are large-grain parallel computations for distributed memory MIMD architectures. We use the following terminology: A *parallel, distributed computation* is a collection of "units" to be distributed over the various processors of a distributed memory architecture, running simultaneously, and exchanging information and synchronizing via *message passing.* Again, the units of distribution are referred to as processes. We assume that the supporting system for such computations includes features to spawn new processes, as well as some message passing primitives [Sha84].

Performance of Parallel, Distributed Computations. Performance of parallel computations is often measured by the ratio of the time to execute "the best" sequential version of the computation to the time to execute the same parallelized computation. For systems with N individual processors this *speedup* has an upper bound of N. This is, however, an optimistic upper bound. Amdahl's law stipulates that, if a parallel computation is made of N units that run in parallel for $(1 - f)$ fraction of the time, and one unit that is active for the remaining fraction f of the time, then the speedup that can be attained is $1/[f + (1 - f)/N]$ (e.g., with $N = 1,000$ and $f = 0.01$, the speedup is 91) [Amd67]. Even small amounts of serial operation (e.g., that caused by synchronization), can drastically affect the speedup. Therefore, partitioning and communication strategies for parallel, distributed computations must attempt to minimize the fraction of serial operation to maximize performance.

Some of the factors involved in choosing a partitioning and communication strategy for a parallel, distributed computation include the "amount of computation" per synchronization (its *granularity*), the nature of the communication and synchronization mechanism, and the identification of specific function to place in a process, (e.g., partitioning by data or functional criteria). The partitioning and communication strategy also has direct consequences on how a parallel, distributed computation will perform if the various processors of the supporting distributed hardware environment are unevenly loaded, or operate at different speeds. Choosing a strategy to obtain a computation that yields high performance is a difficult problem. While some aspects of the partitioning problem can be handled algorithmically for certain classes of algorithms (e.g., see [HHR87]), there currently are no general solutions to this problem. Instead, it appears that the best hope for improved partitioning technology is to provide tools to support the software engineer as he designs the computation.

VISA. *VISA* (VISual Assistant) is such a tool, designed to support prototyping and simulation of parallel, distributed computations. VISA has a graphical interface that is based on *ParaDiGM* (Parallel, Distributed computation Graph Model). ParaDiGM is a modeling technique for representing parallel, distributed computations. It can be used during the design process to describe parts or all of the application under study and to compare possible alternative approaches. We shall see in Section 5 that there are other software models; however, none of these models were designed to support the parallel, distributed model of computation. ParaDiGM has been designed explicitly to model these computations. ParaDiGM graphs are used as a framework for prototyping and simulating software with VISA. In particular, VISA supports the editing of ParaDiGM graphs to describe the computations of interest, and the animation of these graphs to provide visual feedback during simulations. The graphs are supplemented with various attributes, simulation parameters, and *interpretations* (procedures that can be executed by VISA).

This paper is organized as follow. In the next section, we introduce an example that will serve as a case study throughout this paper, and we use it to introduce the ParaDiGM constructs; in Section 3 we describe VISA; in Section 4 we apply VISA to our example, and provide an evaluation of the tool; we discuss related work in Section 5; we offer some conclusions in the final section.

2 Case Study

In this section, we introduce the *chaotic relaxation computation.* Based on this example, we describe ParaDiGM, the modeling technique used by VISA. We present a variant of the example, the *synchronized computation.* Both computations use alternative partitioning and communication strategies for the same problem and are used in section 4 to illustrate VISA.

2.1 The Chaotic Relaxation Computation

Chaotic relaxation is an algorithm for solving a system of n equations and n unknowns, $AX = B$ [CM69]. The general idea is that for a particular class of systems of equations, it is possible to repeatedly solve the system for each $X[k]$ simultaneously, using the current best guesses at the values of $X[j]$, where $k \neq j$. The algorithm therefore has a straightforward parallel, distributed implementation. For certain classes of equations, the algorithm will converge.

The computation that implements the approach typically determines the dimension of the system of equations, n; it then reads the values for the matrix A and the vector B into its memory and spawns n identical *worker* processes. As the computation proceeds, the creating process (called the *parent* process) keeps track of the successive values of X, and tests them to decide whether a solution has been reached or not. When the error in the solution is judged to be sufficiently small, the parent process tells the n worker processes to halt. When all processes have halted, the parent process saves the result and terminates.

Each worker process is given a copy of a row of A (e.g., row k), and the corresponding element of B ($B[k]$). The worker process first obtains a copy of the current best-guess at the value of the solution vector, X, and then it computes $X[k]$ from the guess vector X, the row of A and the value of B it knows. After computing $X[k]$ the worker process sends the value of $X[k]$ to the parent process which updates the value of the current best-guess at X. In return, if the halting test was negative the parent process sends the current value of X to the worker, which immediately starts a new iteration. Otherwise the parent process returns a termination message that causes the worker process to terminate.

2.2 Modeling Chaotic Relaxation with ParaDiGM

ParaDiGM is composed from two submodels: PAM (Process Architecture Model) that describes the processes involved in the computation and the pattern of communication among them; and DCPG (Distributed Computation Precedence Graph) model to provide a more detailed view of the computation. Both PAM and DCPG graphs are generated from a set of nodes and arcs (see Figure 1 for the various node and arc types).

Figure 2 is a ParaDiGM graph for the chaotic relaxation computation. This computation involves a *parent* process and a number of *worker* processes. The top level graph is a PAM graph. In this graph, the node labeled "1" represents the parent process, and the nodes labeled "2" and "3" represent two of the worker processes. The square nodes labeled "4" and "5" illustrate that the parent process communicates on a point-to-point basis with each of the worker processes (2 and 3). The ellipse node indicates that while the model represents only two of the worker processes, there may be more.

The bottom level graph in Figure 2 is a DCPG graph. It is a precedence graph that shows the various tasks and messages involved in the computation. The boxed labels indicate a mapping between the nodes in the PAM and DCPG graphs. The unboxed labels are identifiers for DCPG nodes where a letter indicates the type of the node (c for control node, m for message node, s for selective node and e for extension node). The nodes are also labeled with small text indicating the nature of the task or data structure they model. The plain arcs indicate control flow, and the bold arcs message flow. The bold dashed arcs indicate access to local data.

In this graph, there are three threads of control: All the nodes in the first thread (c1, c2, c3, c4, c5, c6) are labeled with a boxed 1 because they correspond to the parent process also labeled 1. The nodes in the second thread are c7, c8, c9, and c10 corresponding to the worker process labeled 2. The third thread of control (nodes c11, c12, c13 and c14) also maps onto a worker process (labeled 3). The ellipse node labeled c15 indicates that while the model represents only two workers, there may be more.

Node c1 is a *start* node, which indicates the initial control flow point. Nodes c6, c10, and c14 are *end* nodes which model the termination of processes. Node c2 is a *spawn* node, that models the creation of a number of worker processes. Nodes c3, c4, c7, c9, c11, and c13 are *task* nodes which serve as a general representation for tasks. Tasks that involve some testing and "branching" are

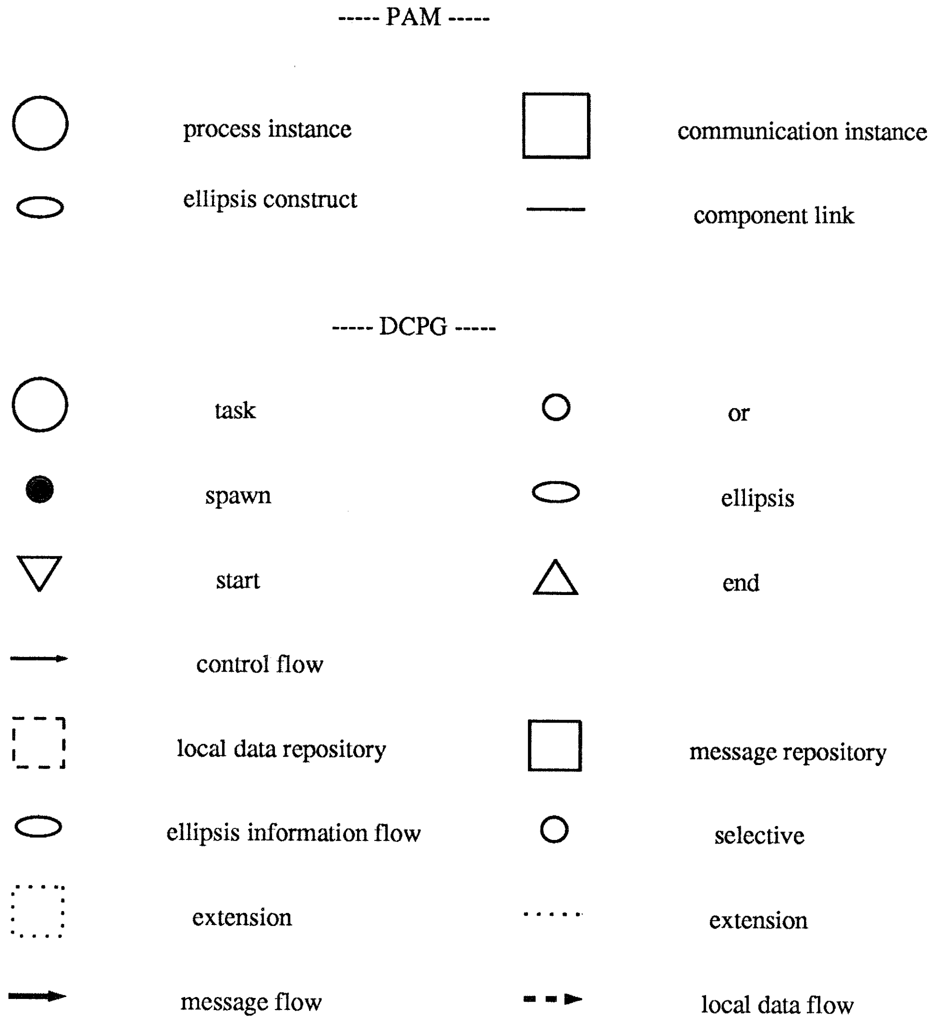


Figure 1: Legend for the ParaDiGM Graphs

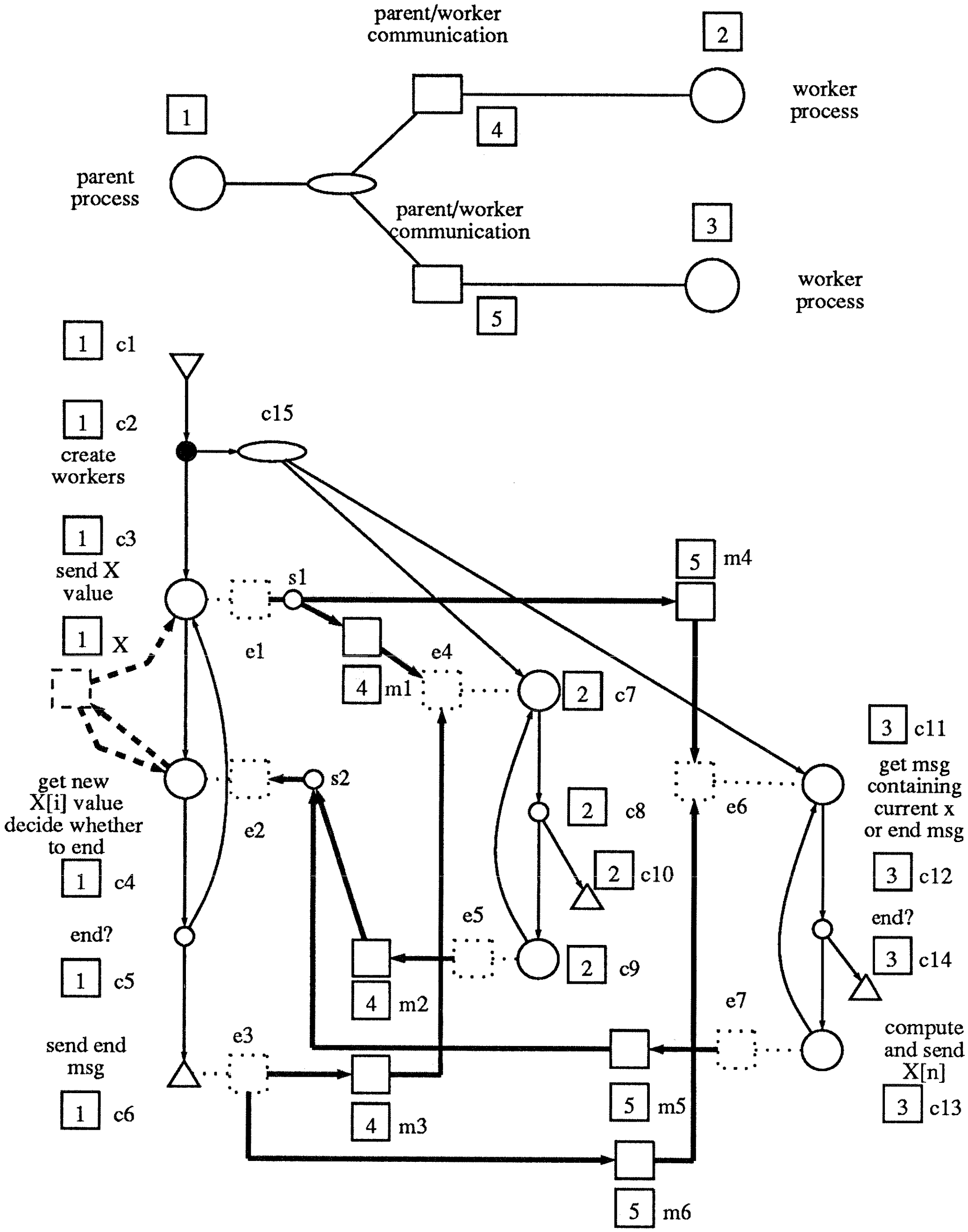


Figure 2: ParaDiGM Model for Chaotic Relaxation

represented by *or* nodes such as c5, c8, and c12. The node labeled X models the vector X, a data structure local to the parent process that is read by nodes c3 and c4, and updated by node c4.

The task modeled by c3 also sends messages to the worker processes. This is modeled by the construct made of nodes s1 (*selective node*) and nodes m1 and m4 (*message nodes*). This construct indicates that the task modeled by c3 sends a message to one of the worker processes. The construct modeled by s2, m3, and m6, indicates that c4 receives a message from one of the worker processes. Notice that nodes m1, m2, and m3 all model messages between the parent process and the worker process labeled 2, which correspond to the communication labeled with a boxed 4. Similarly, nodes m4, m5, and m6 all map to the communication node labeled with a boxed 5.

Finally, the dotted nodes labeled e1, e2, e3, e4, e5, e6 and e7 are what we call *extension* nodes, and are used to separate the control flow from the message flow in the graph. They are used as an "extension" to the node they are linked to by a dotted arc.

A complete description of the model can be found in [Dem89]. We have used ParaDiGM to model different kinds of systems and algorithms ranging from client/server operation, to accounting practices used in a PBX, to various versions of an adaptive global optimization algorithm [DSN88]. ParaDiGM is intended to represent parallel, distributed computations in terms of the functionality, process partitioning, interprocess communication, synchronization, control tasks, and message and data flow, all in a single framework. ParaDiGM representations have proved to be concise and precise. Their visual nature makes them ideal candidates for serving as a basis for a software tool using a graphical user interface.

2.3 A Variant: the Synchronized Computation

In the chaotic relaxation algorithm, when a worker process is done computing $X[k]$, it obtains the current values of X from the parent process and immediately starts a new iteration. A possible alternative is to have all the worker processes synchronize after each iteration, and obtain a whole new set of values for X before they start a new iteration. We call this algorithm the *synchronized algorithm.*

The computations that implement the two algorithms are very similar. They differ only in the way the participant processes synchronize, i.e. they are alternative partitioning and communication strategies for the same problem. We shall use them in section 4 to illustrate the use of VISA for comparing alternative partitioning and communication strategies for a computation.

3 VISA

VISA is a prototyping and simulation tool based on the ParaDiGM model. It can be used to prototype computations and to simulate their execution under various conditions. VISA was developed in a Sun Workstation environment, using Sun graphics and network facilities. It reuses parts of the Olympus Modeling System [Nut89], a companion study on graph model interpreting systems, although VISA and Olympus support different models.

To illustrate how VISA works, we go through a typical VISA session. Figure 3 shows a simple DCPG graph modeling such a session. VISA users describe a computation by creating ParaDiGM graphs, with interpretations and simulation information in textual form. The interpretations are sequences of code to be executed when the prototype of the computation is run; simulation information includes distribution functions to be used when execution time values are needed, and load factors for representing the effective speed of a processor. VISA users can view animations of the graphs as the simulations are run. Summary results are available when a simulation ends. We now describe the different functions available through VISA, in turn.

3.1 Editing the graphs

The editor supports the creation and modification of a PAM graph and a DCPG graph. Both graphs are edited in the same window using different sets of commands (see Figure 4). In addition to the graphical information, the user can place labels on the graph components.

3.2 Interpretation Language

As in the Olympus system [Nut89], an interpretation procedure can be associated with each DCPG control node. The interpretation language is C supplemented by a library of functions that contains primitives to describe parallel, distributed computations (matching the ParaDiGM graphical constructs). These primitives include the *sys_spawn* primitive to create a single process, the *sys_spawn_ellipsis* to create a collection of processes (to match the DCPG ellipsis construct), the *sys_send* primitive to send a message, and the *sys_receive_block* and *sys_receive_noblock* primitives

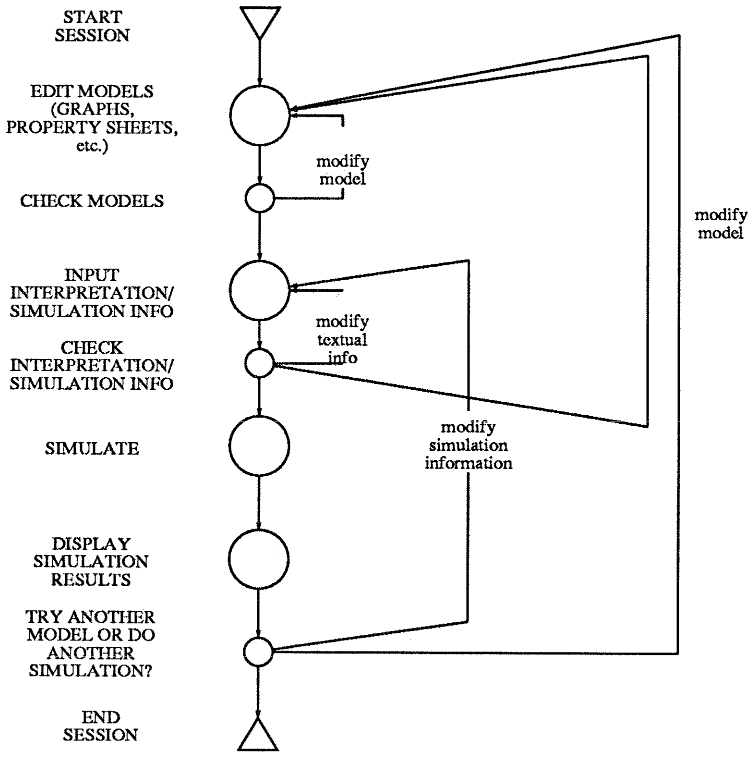
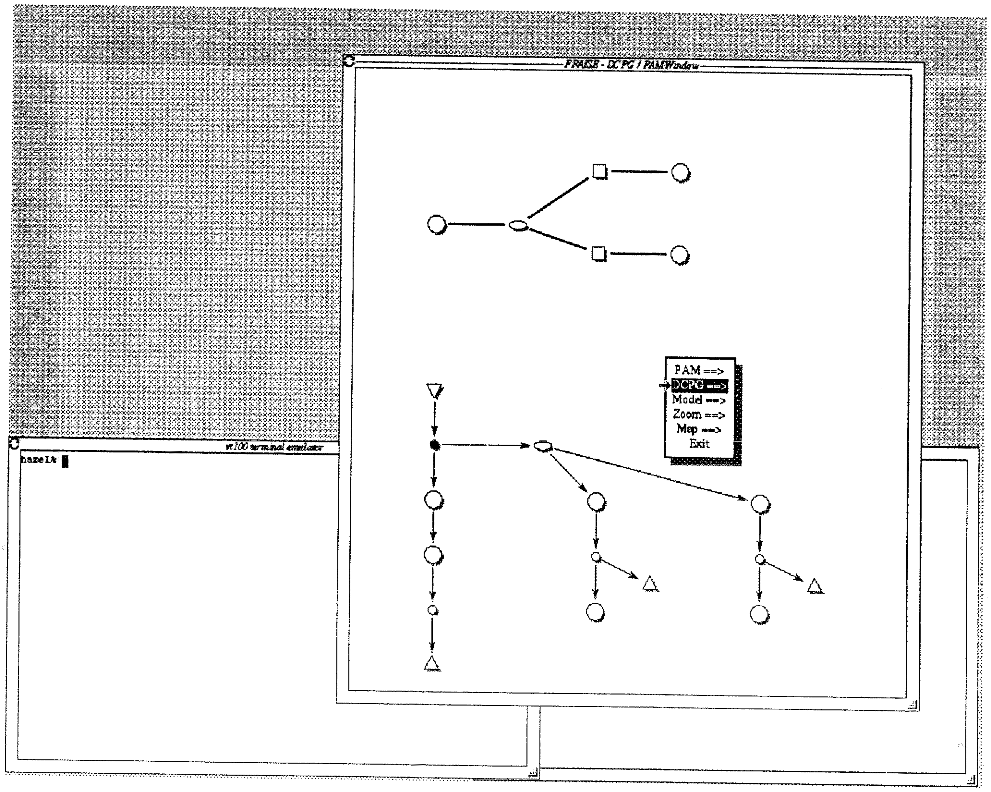

START SESSION
EDIT MODELS (GRAPHS, PROPERTY SHEETS, etc.)
CHECK MODELS
INPUT INTERPRETATION/ SIMULATION INFO
CHECK INTERPRETATION/ SIMULATION INFO
SIMULATE
DISPLAY SIMULATION RESULTS
TRY ANOTHER MODEL OR DO ANOTHER SIMULATION?
END SESSION
modify model
modify textual info
modify simulation information
modify model

Figure 3: A Typical VISA Session



```
hazel%
```

Figure 4: Example of an Editing Session

to receive messages. (See [Dem89] for a more complete description of the interpretation language.) Figure 5 shows an example of an interpretation procedure for node c4 of the parent process.

The interpretation procedures provide additional information when the graphical information is not sufficient to model the computation (e.g., to specify the semantics of an *or* node, or the number of processes to be created in an ellipsis construct). The interpretations are executed during the simulation of the computation; they provide a way to prototype the computation. The user may write simple C procedures to simulate aspects of a computation, or detailed procedures that can eventually be reused when the computation is implemented. VISA supports parallel execution of several interpretation procedures.

3.3 Simulating

Serial software execution times can be used to specify process execution times; however, standard Sun UNIX primitives are insufficient to measure the execution time of a piece of software to obtain the desired time values. For this reason, the simulator uses timing information provided by the user in the form of distribution functions and time values in tables.

There can be several instances of the "same process" (e.g. several worker processes); they are instances of the same *process class.* Similarly, identical communication relations (involving processes of the same class, following the same pattern of communication) are members of the same *communication class.* The simulation information for the PAM graph, is specified for each class. For each process class, the parameters are a table of load factors, a table of time values, and a distribution function. For each communication class, the parameter is a table of delays. A distribution function may also be specified for each DCPG control node.

The simulator infers execution times from relevant DCPG node distribution functions if they are specified; otherwise, it attempts to obtain a value from the corresponding PAM class. If no distribution function was specified, a value from the "time value" table is used (the table is used circularly: the index of the current element in the table is given by the number of events for which the table has been used, modulo the number of elements in the table). Notice that while different samplings of a distribution function yield different values, a simulation done with time values only is deterministic, and therefore yields the exact same results when reproduced.

Computing the Simulation Times. Since VISA combines the execution of interpretations

```
#define PRECISION     0.01
#define NUM_WORKERS   3
#define MSG_SIZE      32

/* globals */
float   X[NUM_WORKERS];
short done[NUM_WORKERS];

/* some code deleted */

proc4()
/*
 * Parent Process: task node
 * wait for a new X[k] value from one of the workers, get it, and
 * set up flag for stopping condition if relevant
 */
{
        char msg[MSG_SIZE];
        short msg_length;
        float result, diff, fabs();

        /* block waiting for a message from one of the workers */
        sys_receive_block(msg, &msg_length, &num, 0);
        sscanf(msg, "%f", &result);


        /* test for stopping condition */
        diff = fabs(result - X[num]);
        if (diff <= PRECISION)
           done[num] = TRUE;
        X[num] = result;

        task_return(0, 0);
}
```

Figure 5: Example of interpretation procedure

with the evaluation of simulation times, several problems can occur: (1) the estimated simulation time can be smaller than the execution time; (2) the events can get out of order because of the relative loads of the processors on which the various processes of VISA are executing; and (3) the simulator introduces some overhead and might therefore cause some delay in the simulation. In order to maintain relative ordering of the events, we introduce a scaling factor by which all the times are multiplied (which is equivalent to slowing down the simulation by that factor).

Let us introduce some notations:

- $T0$: start time for simulation;
- $EI(e)$: interval of time it takes for the simulator to execute the operations associated with event e;
- $SI(e)$: estimated time associated with the event e to simulate the execution of the event e;
- $ASI(e)$: adjusted simulation time for the execution of the event e (time the simulator will actually take to execute the event);
- SF: scaling factor.

By definition:

$$ASI(e) = SI(e) * SF; \qquad (1)$$

The aim is to choose SF such that for all events e: (1) $ASI(e) \geq EI(e)$, (2) SF is big enough so that the overhead due to the simulator is negligible compared to the adjusted simulation times, and that the actual relative loads of the processors can be ignored. Notice that the choice of SF depends on the typical real execution times and simulated execution times. We let the user pick a value for SF. The simulator detects when the scaling factor is too small (when it gets events after their adjusted simulation time is passed), and gives feedback to the user who can then adjust it.

Simulating Processor Load. The same idea of time adjustment can be used to simulate the relative loads or speeds of processors. This is done by assigning a load factor to each PAM component. The bigger the load factor, the slower the processor running the PAM component (either because of a heavy load or because the processor is slower). So if PLF designates the Processor Load Factor, we have:

$$ASI(e) = SI(e) * SF * PLF; \qquad (2)$$

A similar idea is used to simulate transmission times.

Using the Simulation Times. The simulator is notified of all the events. It computes the adjusted simulation time interval associated with each event and deduces the time at which the event should occur from the simulation point of view. It queues the events in order (with respect to the computed time), keeps track of the simulation time, checks the queue at regular intervals, and executes the events in the queue for which the time is less than or equal to the current time.

If the time computed for an event is less than the current time (incorrect ordering of events), the simulator notifies the user of the system (who can then take action by adjusting the scaling factor SF).

See [Dem89] for an extended discussion of simulation with VISA.

3.4 Animating

Once the simulation information is specified, the simulation can start. During the simulation the user is provided with visual feedback in the form of tokens flowing through the DCPG graph, in the style of tokens used in Petri Nets [Pet81]. The simulator also monitors the number of processes created, terminated, and idle, and displays a message at regular intervals to update these values.

3.5 Collecting Results

The simulator collects information about a simulation as it runs and generates a report when the simulation terminates. This information is available to the user at the end of the simulation. The information collected is of three types: global information, per-process information, and per-DCPG-node information. The global information includes:

- the number of processes created;
- the number of idle/active/terminated processes at the time the simulation was ended;
- the number of messages sent/the number of receives performed;
- the number of spawns performed;
- the number of times events were executed out of order (greater than zero if the scaling factor was not big enough).

The per-process information is collected for each of the spawned processes (whether represented on the ParaDiGM graphs or not). It includes:

- the process start time/end time/duration;
- percentage of idle time (waiting for messages/for spawns);
- the number of messages sent/number of received performed;
- the number of spawned performed.

The per-DCPG-node information is collected for each DCPG control node (whether part of a process represented on the ParaDiGM graphs or not). It includes:

- the number of times executed (a node is executed several times if it is part of a loop);
- average execution time/average idle time.

4 Experiments with the Tool

In this section, we present the results of two simulations of each of the computations presented in Section 2. We use them to illustrate the utility of VISA. The two computations correspond to alternative partitioning and communication strategies for the same problem.

4.1 Experiments with the Example Computations

We ran two tests with each of the computations. In all four tests, the time values were obtained from "time value" tables. The interpretation code used in these tests is a prototype of the computations. Although the interpretation yields the solution to the system of linear equations, due to the support provided by VISA it requires significantly less effort than implementing the full computation, scheduling it on processors, and running it under various conditions. In other tests (see [Dem89]), the interpretation is more of a simulation of the computation than an actual implementation.

Each of the four tests was run on a 3 by 3 matrix, with an initial condition such that the algorithm would converge. In these tests, we were interested in predicting the performance of the two computations under various load conditions. In the first test, the three worker processes were simulated as running on equivalently loaded processors. In the second test, one of the worker processes was simulated running on a processor that is one third as fast.

The results of the four tests are summarized in Table I. For each of the tests, and for each process, we recorded the number of iterations needed for the algorithm to converge, the duration of the process, and the percentage of time the process spent waiting for messages to arrive.

Results with the chaotic relaxation computation. The first and third columns in Table I correspond to the tests for the chaotic relaxation computation. When one of the workers is one third as fast, the computation lasts 50 percent longer (as indicated by the duration of the parent, and the workers) than when they all run at the same speed; the two faster workers also need to do about two times as many iterations since they do not get an updated value of X[3] from the slower worker as often as they would if all workers ran at the same speed. The percentage of time spent idle is divided by two for the parent and the two faster workers, since they do about twice as many iterations in a time that is only fifty percent longer.

Results with the synchronized computation. The second and fourth columns correspond to the tests for the synchronized computation. In this case, the computation lasts 100 percent longer when one of the workers is three times slower, than when they all run at the same speed. The number of iterations is the same, since all the processes synchronize after each iteration. However, the percentage of time spent waiting for messages for the parent and the two faster workers is three times larger than when the three workers are run at the same speed.

Comparison of the results for the two computations. The first two columns of this table indicate that the synchronized computation performs slightly better than the chaotic one if all the processes run at the same speed. In the synchronized version, the parent does not have as much computing to do as in the chaotic version: the parent waits for all three workers to complete an iteration before it itself iterates. In the chaotic version, the parent iterates each time any worker completes an iteration. The last two columns indicate that the chaotic relaxation computation performs better than the synchronized computation under uneven load conditions. It does, however, use the CPU more intensively than the synchronized version (since it iterates more in a shorter period of time).

From these experiments, one can conclude that if the load of the processors is equivalent (e.g. there are no other users on the system and all processors operate at the same speed), one should

Table I: Comparison of Results for Four Tests

Process	data	bal chaotic	bal synchro	unbal chaotic num4 loaded	unbal synchro num4 loaded
parent num 1	num iterations	18	6	29	6
	duration (ms)	56	42	84	96
	% time idle (msg)	2	14	1	62
worker num 2	num iterations	7	6	13	6
	duration (ms)	55	41	83	95
	% time idle (msg)	29	17	16	64
worker num 3	num iterations	7	6	13	6
	duration (ms)	55	40	83	94
	% time idle (msg)	30	14	17	63
worker num 4	num iterations	6	6	5	6
	duration (ms)	52	38	87	102
	% time idle (msg)	35	11	0	0

prefer the synchronized computation over the chaotic one. Otherwise, one should prefer the chaotic computation.

These two tests are simple, but they illustrate the nature of the results that can be produced by VISA.

4.2 Evaluation of the Tool

We have presented simulation results for two computation that correspond to alternative partitioning and communication strategies for the same problem. Although these computations are simple, VISA still proves to be useful: it allows the programmer to verify his intuition, and to support it with measurements, without having to fully implement alternative versions of the computation under study, and to run those computations under various conditions. The interpretations developed for the example computations are quick prototypes of the corresponding computations. The measurements made on the synchronized computation and on the chaotic one support the intuition that the chaotic relaxation computation should react more gracefully to load imbalance. In more complex cases, the same kinds of tests would help predict the behavior of the computation.

Although using VISA may appear to generate overhead in developing parallel, distributed computations, we believe that less total project time will be incurred with its use. It has been our experience that through VISA one can quickly design and prototype an implementation of a parallel, distributed computation, become familiar with it by animating it, evaluate its performance characteristics by running simulations of it under various load conditions, and try variants of it, without the cost of actually implementing it. The programmer can focus on such things as the way an implementation will react to load imbalance, the efficiency with which the processors are used by a computation, and the cost of the communication and synchronization for a given implementation. VISA is therefore useful for predicting the performance of a parallel, distributed computation during its design phase. The language used to prototype the computation is C with a supplementary library and the interpretations written for the purpose of the model can be reused for the actual implementation of the computation. Finally, the graphs given as input to VISA can be used as documentation for the computation under study.

5 Related Work

With the technological advances that have been made in high-resolution and bit-mapped graphics, many software tools now use a graphic interface to a modeling or prototyping environment.

For example, STATEMATE [HLN*88] is a CASE (Computer Aided Software Engineering) system that incorporates the Statechart [Har87] mechanism developed by Harel. Statecharts are an extension of state diagrams used for studying the behavior of systems. In STATEMATE, statecharts are used as the controlling mechanism for activity charts which describe the functional characteristics of a system and are complemented by module charts which allocate the functionality to system components. The charts are supported by a graphical editor, a data dictionary, and a generalized query capability. Although the approach in STATEMATE is similar to ours, it is meant for reactive systems, and is not well suited to the parallel, distributed computations we focus on in this work.

SARA (System ARchitects Apprentice) [EFRV86], is another CASE system. In SARA, the structure of a system is represented as a hierarchy of modules connected via sockets and interconnections. The behavior of the system is modeled using GMB (Graph Model of Behavior). GMB consists of a control flow graph very similar to Petri Nets, a data flow graph, and a description of the processes associated with the nodes of the data flow graph. SARA is well suited for the design of concurrent real-time systems, but could not very naturally support the design of parallel, distributed computations, as it does not have the appropriate primitives built into the system.

Finally, PARET (Parallel Architecture Research and Evaluation Tool) [NE88] is described by its authors as an environment for the study of interaction of algorithms and architectures. The underlying model is a directed flow graph. Through PARET, the effect of varying physical resources on system performance and alternate mapping, scheduling, and routing strategies can be studied. Interactive simulation, run-time measures, and summary statistics are supported. PARET is mostly targeted at modeling and studying architectural aspects as opposed to the algorithmic aspects on which we are focusing.

There are many other systems we could describe, such as CAPS [Luq89] a rapid prototyping system that relies on the PSDL(Prototype System Description Language) model. None of these systems, however, can naturally support the design of distributed computations: STATEMATE is meant for reactive systems, SARA is well suited for the design of concurrent real-time systems, PARET focuses on architecture aspects, and CAPS is general-purpose but focused on real time systems. VISA, on the other hand, is based on the ParaDiGM model that specifically addresses the issues that are inherent to parallel, distributed computations.

Finally, since part of the originality of VISA lies in the model on which it is based, let us provide some background for ParaDiGM. There have been a number of different graph models of software over the last twenty years, e.g., see [Bae73] and more recently, [Bro87]. The immediate predecessor of the DCPG model is the BPG (Bilogic Precedence Graph) model [Nut87] (which, in turn, is descended from Petri nets [Pet81]); while BPGs were designed for general purpose modeling of parallel computations (see [NBD*89]), DCPGs focus on large-grained, message-based local computations. The basic PAM graph is most similar to communication graphs such as the one used in the POKER system [Sny84].

6 Conclusion

The performance of a parallel, distributed computation is very dependent on the partitioning and communication strategy that is chosen to implement it. Such a strategy determines the functionality implemented by each participant process, and the nature and amount of communication and synchronization that will have to take place among the participant processes. It therefore determines how a computation will react to load imbalance or differences in speed of the processors of the supporting distributed hardware environment, and thus the efficiency with which these processors will be used.

There are no known general conditions to guarantee that partitioning and communication strategy for a computation will yield high performance. VISA is a prototyping and simulation tool intended to aid the designer of parallel, distributed computations in manually partitioning the computation, with rapid feedback on the behavior of a strategy.

VISA makes it easy to experiment with different implementations under various load conditions and with various input data. In these experiments, the designer can choose to focus on specific parts of the computation and to simulate others. The interpretation code can eventually be reused for the final implementation.

VISA is built on ParaDiGM, a graph model intended to represent distributed computations in terms of functionality, process partitioning, interprocess communication, synchronization, control tasks, and information flow. Part of the originality and the strength of VISA results from the fact that ParaDiGM specifically addresses the issues that are inherent to parallel, distributed computations. In addition, ParaDiGM yields visual representations of computations which makes it an ideal candidate for serving as a basis for a software tool using a graphical user interface.

In this paper, we have given a brief introduction to some of the ParaDiGM constructs; we have described the main functions of the VISA tool; finally, we have illustrated the use of VISA, obtaining simulation results for two implementations of a simple algorithm for solving a system of linear equations.

Experiments with the current version of VISA have revealed several ways in which the tool could be improved. In particular, they include user interface issues, issues related to the efficiency of the simulator, as well as ideas for additional functions (e.g., a function that would check that ParaDiGM graphs are correctly built). One of the directions of our continuing research therefore is to refine VISA.

We have used VISA to study computations such as the chaotic relaxation computation presented here and several variants of the client/server model (see [Dem89]). Another direction of the continuing research is to do experiments with more complex algorithms (e.g. Adaptive Global Optimization Algorithm, [DSN88]), to use VISA to guide the partitioning and communication strategy for such computations, and to predict their performance.

7 Acknowledgements

We are grateful to Adam Beguelin, Jeff McWhirter, Mike Schwartz, and Dave Wagner, whose feedback helped us in the preparation of the final version of this paper.

References

[Amd67] G. M. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, 1967.

[Bae73] J. L. Baer. A Survey of Some Theoretical Aspects of Multiprocessing x. *Computing Surveys*, 5(1):31–79, March 1973.

[Bro87] J. C. Browne. A Unified Approach to Parallel Programming. June 1987. 1987 Summer Workshop in Parallel Computation - University of Colorado, Boulder.

[CM69] D. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[Dem89] I. M. Demeure. *A Model, ParaDiGM, and a Software Tool, VISA, for the Representation, Design and Simulation of Parallel, Distributed Computations.* PhD thesis, University of Colorado, Boulder, August 1989.

[DSN88] I. M. Demeure, S. L. Smith, and G. J. Nutt. *Modeling Parallel, Distributed Computations Using ParaDiGM - A case study: the Adaptive Global Optimization Algorithm.* Technical Report CU-CS-419-88, Department of Computer Science - University of Colorado, Boulder, December 1988. To Appear in the Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing - December 11-13 1989.

[EFRV86] G. Estrin, R.S. Fenchel, R. R. Razouk, and M.K. Vernon. SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, Feb 1986.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming - North-Holland*, 8:231–274, 1987.

[HHR87] Catherine E. Houstis, Elias N. Houstis, and John R. Rice. Partitioning pde computations: methods and performance evaluation. *Parallel Computing*, 5(1I&2):141–163, July 1987.

[HLN*88] D. Harel, H. Lackover, A. Naamad, A. Pnuelli, M. Politi, R. Sherman, and A. Shtul-Tauring. STATEMATE: A Working Environment for the Development of Complex, Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering*, 1988.

[Luq89] Luqi. Software Evolution Through Rapid Prototyping . *Computer (IEEE)*, 13–25, May 1989.

[NBD*89] G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter, and B. Sanders. Olyumpus: An Interactive Simulation System. In *1989 Winter Simulation Conference*, Washington, D.C., December 1989.

[NE88] K. M. Nichols and J. T Edmark. Modeling Multicomputer Systems with PARET. *Computer (IEEE)*, 39–48, May 1988.

[Nut87] G. J. Nutt. *Bilogic Precedence Graph Models.* Technical Report CU-CS-363-87, Department of Computer Science - University of Colorado, Boulder, May 1987.

[Nut89] G. J. Nutt. A Flexible, Distributed Simulation System. In *10th International Conference on Application and Theory of Petri Nets*, pages 210–226, Bonn, West Germany, June 1989.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., 1981.

[Sha84] S.M. Shatz. Communication Mechanisms for Programming Distributed Systems. *Computer (IEEE)*, 21–28, June 1984.

[Sny84] L. Snyder. Parallel Programming and the Poker Programming Environment. *Computer (IEEE)*, 27–36, July 1984.